Prova Finale
Progetto Reti Logiche

Lucia Famoso - (Codice Persona 10684237)

Agosto 2022

Indice

1 Introduzione

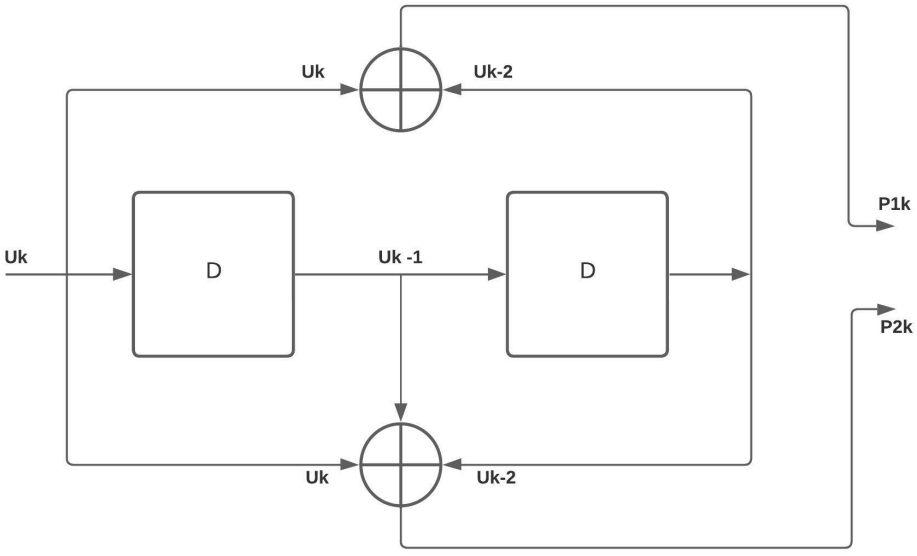
1.1 Riassunto delle Specifiche

Lo scopo del progetto finale di reti logiche 2021/2022 è stato di implementare un modulo hardware in VHDL che, data in input una sequenza di **W** parole da 8 bit, lunga massimo 255byte, restituisse in output, tramite l'applicazione di un flusso convoluzionale con tasso di trasmissione 1/2, una sequenza di **Z=2*W** parole da 8 bit.

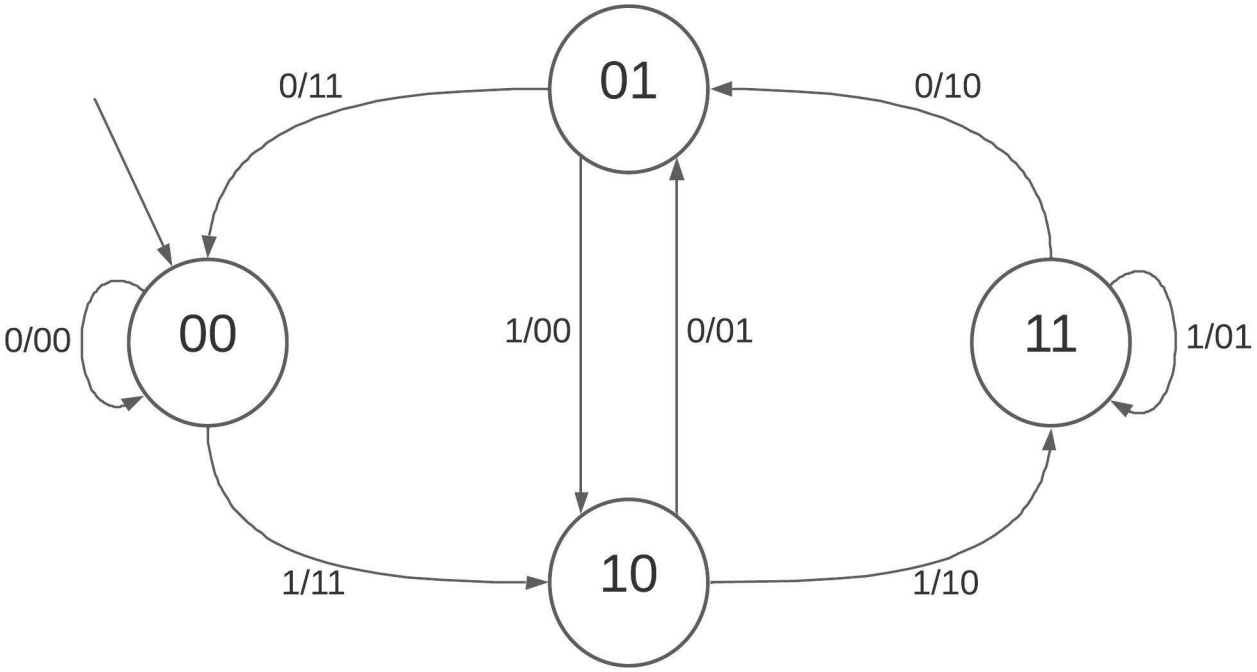
1.2 Struttura del Convoluttore

Il convoluttore è la componente del modulo che si occupa della codifica e dell'elaborazione delle parole in input.
Il suo funzionamento prevede, per ogni bit **Uk** della sequenza serializzata U, acquisito da input, l'emissione in output di due bit **Pk1** e **Pk2** concatenati in tale ordine: **Uk/Pk1Pk2**. Il flusso continuo **Yk** generato dalla concatenazione di tali bit, una volta parallelizzato, corrisponde alla sequenza **Z** desiderata.

Il convoluttore si serve di due flip flop di tipo D e altrettante porte XOR per la computazione, posizionati secondo lo schema sottostante:



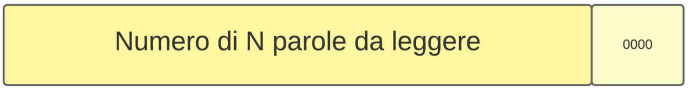
La procedura logica attuata dal convoluttore può, inoltre, essere rappresentata tramite il seguente diagramma degli stati, implementato all'interno dello stato **CONVOL_COMPUTING** del modulo:



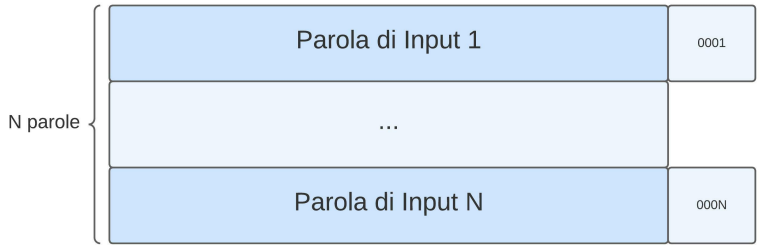
1.3 Struttura della Memoria

La sequenza di **W** parole da 8bit è immagazzinata in una memoria ad indirizzamento al Byte con cui il modulo comunica ed al cui interno i dati sono memorizzati in questo ordine:

- All'indirizzo 0000 è memorizzato il numero totale di parole contenute nella sequenza. Questa cella di memoria viene acceduta una singola volta all'interno di una computazione completa del modulo e il valore contenuto al suo interno viene assegnato al segnale di appoggio *tot_cycle*, il cui significato viene approfondito nella sezione successiva.



- Tra l'indirizzo 0001 e l'indirizzo 000N sono collocate le parole da 8bit della sequenza di ingresso W, posto N il numero di parole totali. Il modulo legge e computa una singola parola per ciclo.



- Tra l'indirizzo 1000 e l'indirizzo 000M saranno, invece, collocate le parole da 8bit della sequenza di uscita Z, con M=2*N. Per ogni parola computata, il modulo ne scriverà due di output in due celle di memoria adiacenti.

1.4 Interfaccia del Componente

L'interfaccia del componente è definita cosi:

```
entity project_reti_logiche is
    port (
    i_clk : in std_logic;
    i_rst : in std_logic;
    i_start : in std_logic;
    i_data : in std_logic_vector(7 downto 0);
    o_address : out std_logic_vector(15 downto 0);
    o_done : out std_logic;
    o_en : out std_logic;
    o_we : out std_logic;
    o_data : out std_logic_vector (7 downto 0)
);
end project_reti_logiche;
```

- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

2 Architettura

In questa sezione vengono approfonditi i segnali di appoggio interni e gli stati del modulo. A seguire viene aperta una sezione di discussione sulle scelte implementative che hanno portato alla realizzazione dell'implementazione definitiva.

2.1 I Segnali

2.1.1 tot_cycle: integer

Segnale che assume il valore del dato letto all'indirizzo 0000 e determina il numero di cicli che dovrà compiere il modulo, e quindi di celle di memoria totali da leggere.
Ad ogni reset del modulo il suo valore torna a 0.

2.1.2 cycle_index: integer

Segnale che svolge la funzione di contatore dei cicli del modulo. Viene incrementato di uno dopo l'assegnazione di *tot_cycle* e successivamente dopo la computazione di ogni dato. Viene inoltre utilizzato nel calcolo dell'indirizzo di scrittura.
Ad ogni reset del modulo il suo valore torna a 0.

2.1.3 i_bit_id: integer

Segnale che rappresenta l'indice del n-esimo bit da mandare in pasto al convoluttore. Dopo ogni computazione del convoluttore viene decrementato di uno (7 downto 0), al fine di computare il successivo bit, per poi essere riportato a 7 una volta raggiunto lo 0.
Ad ogni reset del modulo il suo valore torna a 7.

2.1.4 write_count: integer

Segnale che determina quante scritture sono state svolte per ogni dato computato. Viene portato ad 1 dopo la scrittura in memoria dei primi 8 bit della sequenza di output e viene riportato a 0 dopo la scrittura dei secondi 8 bit della sequenza di output. Ad ogni reset del modulo il suo valore torna a 0.

2.1.5 memo_data: std_logic_vector(7 downto 0)

Segnale di appoggio che assume il valore del dato da computare durante la fase di lettura dell'input **READ_INPUT**, al fine di evitare errori o sovrascrizioni.

2.1.6 elab_out: std_logic_vector(15 downto 0)

Segnale di appoggio che assume il valore dell'output totale dopo la computazione di ogni dato (15 downto 0). Verrà suddiviso in due byte, servendosi del contatore *write_count*, e trascritto in due celle di memoria adiacenti nella fase di scrittura **WRITE_MEMO**.

2.2 Il Modulo e i suoi Stati

2.2.1 Lo stato IDLING

Stato iniziale. Il modulo è in attesa del segnale di *i_start* che determina la lettura del primo elemento di memoria e l'inizio della computazione.

2.2.2 Lo stato READ_MEMO

Stato di lettura della memoria. Il modulo controlla il contatore dei cicli *cycle_index* e in base al suo valore decide come agire.

- $cycle\_index = 0$: Siamo nel primo ciclo. Viene alzato il segnale di lettura, con 0000 indirizzo di lettura.
  Prossimo stato: **WAIT_R_MEMO**;

- $cycle\_index \neq 0, cycle\_index \neq tot\_cycle + 1$: Siamo in un ciclo intermedio. Viene alzato il segnale di lettura con $(0000 + cycle\_index)$ indirizzo di lettura.
  Prossimo stato: **WAIT_R_MEMO**;

- $cycle\_index \neq 0, cycle\_index = tot\_cycle + 1$: Siamo al termine dell'ultimo ciclo. Viene resettato a 0 il contatore dei cicli e viene alzato il segnale di *o_done*.
  Prossimo stato: **DONE**.

2.2.3 Lo stato WAIT_R_MEMO

Stato di attesa. Il modulo attende un ciclo di clock per poter recuperare il dato letto dalla memoria e poi passa allo stato di lettura input **READ_INPUT**.

2.2.4 Lo stato READ_INPUT

Stato di lettura dell'input. Il modulo controlla il contatore dei cicli *cycle_index* e in base al suo valore decide come agire.

- $cycle\_index = 0$: Siamo nel primo ciclo. Il valore letto dalla memoria rappresenta il totale di dati da leggere. Esso viene quindi convertito in intero e caricato in *tot_cycle*.
  Prossimo stato: **READ_MEMO**;

- $cycle\_index \neq 0$: Siamo in un ciclo intermedio. Il valore letto dalla memoria rappresenta una delle parole da computare. Esso viene quindi caricato in *memo_data* al fine di essere processato. *i_bit_id* viene caricato con l'indice del primo bit da processare, ossia 7.
  Prossimo stato: **CONVOL_COMPUTING**;

2.2.5 Lo stato CONVOL_COMPUTING

Stato di computazione del convoluttore. Il dato letto da input viene dato in pasto al convoluttore che lo processa bit per bit riferendosi all'indice mantenuto da *i_bit_id*.
Dopo la computazione del bit di indice *i_bit_id*, il modulo controlla il valore di *i_bit_id* e agisce di conseguenza:

- *i_bit_id* $\neq 0$: La lettura del dato non è completa. *i_bit_id* viene decrementato di uno per continuare la computazione del convoluttore.
  Prossimo stato: **CONVOL_COMPUTING**;

- *i_bit_id* $= 0$: La lettura del dato è completa. *i_bit_id* viene resettato a 7 e il modulo procede con la scrittura.
  Prossimo stato: **WRITE_MEMO**;

2.2.6 Lo stato WRITE_MEMO

Stato di scrittura. Il modulo alza i segnali di enable e di scrittura poi controlla il valore di *write_count* e agisce di conseguenza:

- *write_count* $= 0$: La scrittura del dato non è completa. Vengono scritti i primi 8 bit del dato di output all'indirizzo corretto e *write_count* viene portato a uno.

- *write_count* $\neq 0$: La scrittura del dato è completa. Vengono scritti i secondi 8 bit del dato di output all'indirizzo corretto e *write_count* viene portato a zero.

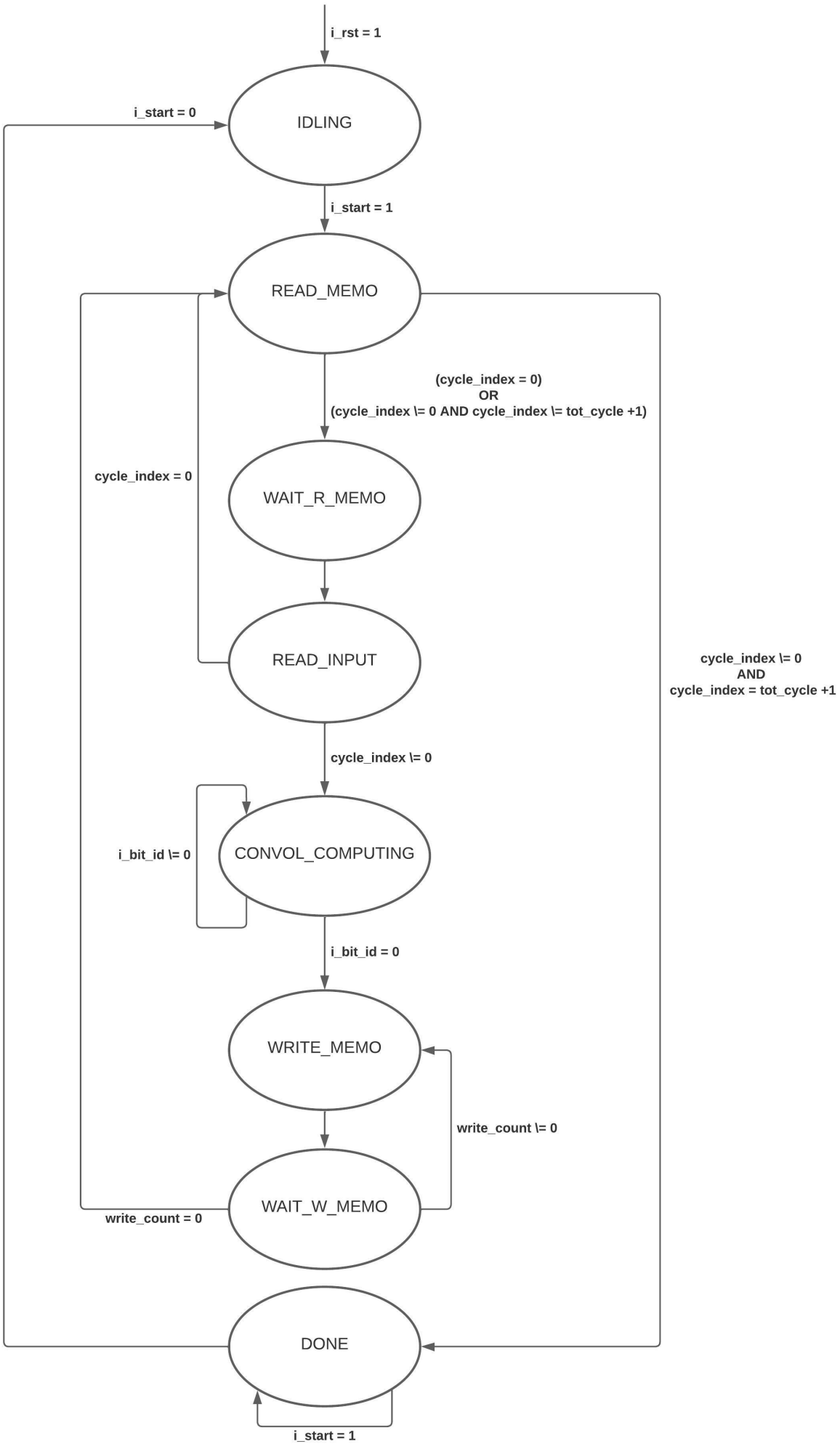
Prossimo stato: **WAIT_W_MEMO**;

2.2.7 Lo stato WAIT_W_MEMO

Stato di attesa. Il modulo attende un ciclo di clock per la scrittura del dato.
In caso sia la seconda scrittura per il ciclo corrente (*write_count* $= 0$), aumenta di uno il *cycle_index* e torna allo stato di lettura **READ_MEMO**. Altrimenti torna allo stato di scrittura **WRITE_MEMO**.

2.2.8 Lo stato DONE

Stato di fine. Tale stato viene mantenuto fintanto che il segnale di *i_start* non si abbassa. A quel punto, viene abbassato il segnale di *o_done* e vengono riportati agli stati iniziali il modulo (**IDLING**) e il convoluttore (**S0**).


i_rst = 1
IDLING
i_start = 0
i_start = 1
READ_MEMO
(cycle_index = 0)
OR
(cycle_index \= 0 AND cycle_index \= tot_cycle +1)
WAIT_R_MEMO
cycle_index = 0
READ_INPUT
cycle_index \= 0
AND
cycle_index = tot_cycle +1
cycle_index \= 0
CONVOL_COMPUTING
i_bit_id \= 0
i_bit_id = 0
WRITE_MEMO
write_count \= 0
WAIT_W_MEMO
write_count = 0
DONE
i_start = 1

2.3 Scelte Implementative

Alcune scelte prese al fine di alleggerire la computazione evitando l'inserimento di più registri e segnali sono state prese in relazione al calcolo degli indirizzi di lettura e scrittura.

- Indirizzo di **lettura**: viene calcolato servendosi del contatore *cycle_index* sfruttando il fatto che l'i-esimo dato da leggere si troverà all'indirizzo 0000 + i.

- Indirizzo di **scrittura**: per gli indirizzi di scrittura il procedimento è stato più complesso a causa del fatto che il primo indirizzo di scrittura fosse 1000 e che per ogni dato letto venissero scritti due Byte e non uno solo.
  Il risultato viene calcolato servendosi comunque del contatore *cycle_index* che questa volta viene moltiplicato per 2. A seconda che si stia scrivento il primo byte o il secondo byte di uscita viene poi sottratto 2 o 1.

Esempio Ciclo 1: $cycle\_index = 1$
Indirizzo di Lettura: $0000 + 1 = 0001$
Indirizzo di Scrittura Byte1: $1000 + (2 * 1 - 2) = 1000 + 0 = 1000$
Indirizzo di Scrittura Byte1: $1000 + (2 * 1 - 1) = 1000 + 1 = 1001$

Un'altra scelta implementativa importante è stata quella riguardante la gestione della scrittura. Una delle opzioni possibili era quella di interrompere la computazione del convoluttore ogni 4 bit letti per scrivere ogni byte di uscita singolarmente.
L'idea è stata scartata per evitare l'aggiunta di più segnali di appoggio (per gli indirizzi, il numero di bit già letti ecc...) e per mantenere la computazione e la progressione degli stati del modulo il più *lineare e pulita* possibile (No eccessivi intrecci logici).

Alla considerazione di linearità fatta poco sopra si lega la scelta di rendere lo stato **READ_MEMO** l'unico in grado di raggiungere lo stato terminale **DONE**. Ciò ha permesso l'utilizzo del registro *cycle_index* come unico e solo discriminante per il termine della computazione.
Di conseguenza, il contatore dei cicli viene incrementato sempre e solo dopo aver terminato la coppia di scritture, oppure dopo aver letto e memorizzato il dato il posizione 0000.

3 Risultati Sperimentali

3.1 Report di Sintesi

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	155	0	0	134600	0.12
LUT as Logic	155	0	0	134600	0.12
LUT as Memory	0	0	0	46200	0.00
Slice Registers	129	0	0	269200	0.05
Register as Flip Flop	129	0	0	269200	0.05
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Il report di utilizzo rivela che il modulo viene correttamente sintetizzato, senza l'inferimento di Latches, e fa uso di 155 LUTs e 129FF.

3.2 Timing

Il report di timing rivela un time slack di 3.916ns. Ciò significa che il modulo riesce a completare con successo l'esecuzione stando all'interno della costraints di 100ns imposta dalle specifiche.

3.3 Simulazioni e Test

Il modulo passa con successo 10 su 10 dei test facenti parte del test bench fornito insieme alle specifiche. (tb_example e tb_esempio_1 sono equivalenti).
La scelta di non implementare custom test è dovuta al già ampio assortimento di tale bench. A seguire un breve riassunto del contenuto:

1. **tb_esempio_1**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui il flusso di parole W sia composto da 2 parole.

2. **tb_esempio_2**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui il flusso di parole W sia composto da 6 parole.

3. **tb_esempio_3**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui il flusso di parole W sia composto da 3 parole.

4. **tb_doppio_uguale**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui venga richiesta una seconda computazione, ossia venga rialzato il segnale di START dopo l'abbassamento del segnale di DONE postumo alla prima computazione.

5. **tb_re_encode** e **tb_tre_bis**: questi test si propongono di verificare il corretto funzionamento del modulo nel caso di tre computazioni successive riferite a tre memorie RAM differenti.

6. **tb_seq_max**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui la lunghezza del flusso W sia massima (255byte).

7. **tb_seq_min**: questo test si propone di verificare il corretto funzionamento del modulo nel caso in cui la lunghezza del flusso W sia minima (0Byte).

8. **tb_reset**: questo test si propone di verificare il corretto funzionamento del modulo nel caso di reset asincrono su una singola simulazione con sequenza di lunghezza 6.

9. **tb_tre_reset**: questo test si propone di verificare il corretto funzionamento del modulo nel caso di reset asincrono sulla prima di 3 simulazioni consecutive.

4 Conclusioni

Avendo programmato sempre codice sequenziale, il primo approccio alla programmazione non sequenziale ammetto esser stato alquanto traumatico. Mi ci sono voluti un paio di giorni per prendere dimestichezza e comprendere i meccanismi e l'importanza dei registri "next", ma una volta presa confidenza lo sviluppo è terminato senza troppi intoppi di percorso. Detto ciò, posso ritenermi molto soddisfatta del lavoro fatto fino a qui e delle nuove conoscenze su VHDL acquisite grazie ad esso.

Scrivere questa relazione mi ha inoltre aperto un mondo sul linguaggio LaTeX e sulle sue possibilità, che di sicuro si riveleranno fondamentali per continuare a documentare correttamente i miei progetti futuri.